3rd Python Workshop Tutorial

Extending and Embedding Python

Day Two – Defining Objects

Guido van Rossum

guido@CNRI.Reston.Va.US
http://www.python.org

Day Two - Overview

- Defining new Python objects in C
- Using "Modulator" to generate extension templates
- Using "BGEN" to generate complete extensions

Defining New Python Objects In C

- object structure
- type object
- allocation
- deallocation
- special operations
- methods
- instance variables
- making a numeric type
- making a sequence type
- making a mapping type

Object Structure

- Each object type has its own struct definition
- Struct starts with required boilerplate:

```
typedef struct {
    PyObject_HEAD
    <your struct members here>
} YourObject;
```

- Macro **`PyObject_Head`** expands to:

```
    int ob_refcount;
    struct _typeobject *ob_type;
```

- (More if you have certain debugging flags on)

Type Object

- Statically initialized structure
- Similar to C++ virtual function vector
  - contains pointers to functions that implement behavior
    - » e.g. deallocation, **`print, getattr(), cmp()`**
  - also contains some strings, e.g. type name and doc string
- Also doubles as a Python-accessible object
  - **`type(x)`** returns the type object for object **`x`**
  - mostly usable for comparing types
- Construction function is *not* included in type object
  - reason: arguments to constructor differ per type

Allocation

- Your module must provide an interface for allocation
- Usually this is exported to Python as a function, e.g.
  - open_socket()
  - create_window()
- Can also be a method of another object, e.g.
  - window.create_menu()
  - socket.accept()
- Responsibilities:
  - allocate memory
  - initialize all fields
- May raise exception if allocation fails

Deallocation

- Deallocator is a function pointer in the type object
- Implicitly called when reference count reaches zero
- MUST NOT RAISE AN EXCEPTION

- Deallocation method must match allocation method
  - usually malloc/free
  - other allocation styles may be used
    - » e.g. integers have their own free list (saves ≥ 4 bytes/obj)

Special Operations

- Functions defining type-specific behavior
- All optional; if NULL pointer:
  - either system default (e.g. cmp(), repr())
  - or "operation not defined" exception (e.g. getattr(), setattr())
- Each has its own prototype
  - first argument is always an object *of this type*
    - » may need cast to convert it but no run-time check

Methods

- Used to implement behavior unique to a type, e.g.:
  - `list.append(), window.resize(), file.read()`
- Methods are not contained directly in the type object
  - (historical mistake: doc strings are unreachable)
- Use getattr() boilerplate:

```
static PyObject *
GetAttr(PyObject *self, char *name) {
    return Py_FindMethod(methods, self, name);
}
```

- Here, **`methods`** is a method table like before
  - but now, **`self`** passed into implementation is non-NULL

Instance Variables

- Accessed through getattr/setattr behaviors
- You get to choose how they are stored, e.g.:
  - in a Python dictionary (flexible, but harder to access from C)
  - as struct members of your own object
  - as computed values
- Standard support exists for "struct members" version:

  ```
  #include "structmember.h"
  ```

  - define a table mapping names to type and offset
  - call PyMember_Get() in your getattr() behavior routine

Making a Numeric Type

- Type object contains optional pointer to **`PyNumberMethods`** structure
- Contains lots of function pointers to implement add, mul, sub, div, mod, etc.
- These function pointers can't be NULL

Making a Sequence Type

- Similar to making a numeric type: **`PySequenceMethods`** structure
- Implement functions:
  - len(), getitem(), getslice()
  - concat(), repeat()
  - setitem(), setslice()
- The assignment function pointers may be NULL
  - to implement immutable sequences (e.g. tuples)

Making a Mapping Type

- Similar to sequence type : **`PyMappingMethods`** structure
- Even simpler set of functions:
  - len()
  - getitem()
  - setitem()

Using "Modulator"

- Interactive tool to generate "boilerplate" code
- You enter module name and function names
- Modulators generates the C code for you
  - You have to edit the generated C file to add semantics
- Modulator is a one-shot tool
  - it can't save the data you entered and continue later
  - but it saves you lots of typing for the boilerplate
- Modulator also generates object definition boilerplate

- Beware: it's not a good tool for learning!

Demo

- Insert interactive demo of Modulator here

BGEN: Beyond Modulator

- BGEN generates more than boilerplate
  - Generates complete extension modules
  - Knows about standard C types
  - Can be taught about other types (and how to map them)
- BGEN can scan header files
  - Turns function prototypes into wrappers
  - Customization possible (and necessary)
- BGEN regenerates when you change its input
- Alas, harder to use
  - no documentation yet
  - not finished yet

BGEN Run Phases

1. Scan header file
   - generating Python script file
2. Import the generated script
   - creating function description objects
3. Generate code
   - generating C module file

BGEN Components

- Header file scanner (needs customization)
  - Generates a Python module read by the next phase
- Exception specifications (pure customization)
- Type definitions (customizable)
- Control module (ought to be generic)
- Generator (generic)
- Support modules (generic)
  - standard types
  - standard buffer types
  - mac specific types

Sample Header File Input "demo.h"

```
/* Sample header file -- input to demoscan.py */

#include <stdio.h>

#define FILEptr FILE *

/* A function we can handle */
extern int system(const char *command);

/* A function we can't handle yet */
extern FILEptr popen(const char *command,
                     const char *mode);
```

Sample Generated Script "demogen.py"

```
# Generated from 'demo.h'

f = Function(int, 'system',
    (stringptr, 'command', InMode),
)
functions.append(f)
```

Sample Control Module "demoscan.py"

```
from scantools import Scanner

def main():
   scanner = MyScanner("demo.h",
                       "demogen.h", "demodefs.py")
   scanner.scan()
   scanner.close()
   import demosupport

class MyScanner(Scanner): ...see next page...

main()
```

Sample Control Module "demoscan.py" (cont'd)

```
class MyScanner(Scanner):

    def initpatterns(self):
        Scanner.initpatterns(self)
        self.head_pat = "^extern "
        self.tail_pat = ";"
        self.type_pat = "extern[ \t\n]+\(<type>[a-zA-Z0-9_ \t]*[a-zA-Z0-9_]\)[ \t\n]+"
        self.whole_pat = self.type_pat + self.name_pat + self.args_pat

    def makeblacklisttypes(self):
        return ["FILEptr"]

    def makerepairinstructions(self):
        return [
            ([('char_ptr', '*', 'InMode')], [('stringptr', '*', '*')]),
            ]
```

Sample Generated Extension "demomodule.c"

```
#include "Python.h"
#include "demo.h"

static PyObject *demo_Error;

static PyObject *demo_system(_self, _args)
    PyObject *_self;
    PyObject *_args;
{
    PyObject *_res = NULL;
    int _rv;
    char* command;
    if (!PyArg_ParseTuple(_args, "s",
                          &command))
        return NULL;
    _rv = system(command);
    _res = Py_BuildValue("i",
                         _rv);
    return _res;
}
```

Sample Generated Extension "demomodule.c" (cont'd)

```
static PyMethodDef demo_methods[] = {
    {"system", (PyCFunction)demo_system, 1,
     "(char* command) -> (int _rv)"},
    {NULL, NULL, 0}
};

void initdemo()
{
    PyObject *m;
    PyObject *d;

    m = Py_InitModule("demo", demo_methods);
    d = PyModule_GetDict(m);
    demo_Error = PyMac_GetOSErrException();
    if (demo_Error == NULL ||
        PyDict_SetItemString(d, "Error", demo_Error) != 0)
          Py_FatalError("can't initialize demo.Error");
}
```

Guessing Argument Transfer Mode

- Arguments are classified as
  - InMode (input only argument)
  - OutMode (output only argument)
  - InOutMode (input and output argument)
- Scanner employs heuristics, e.g.
  - arguments of type "const char *" are InMode
  - arguments whose name begins with "return_" are OutMode
- Scanner needs help in most cases
  - every library has different conventions

Return Values

- Output arguments are translated to a return tuple:
  - `int func(int a, int b, int *return_a, *return_b)`
    is translated into `func(a, b) -> (x, a, b)`
  - `void func(int *return_a)`
    is translated into `func() -> a`

- Input-output arguments are counted twice
  - once as input argument
  - once as output argument
  - `void func(/*inout*/int *a)`
    translates into `func(a) -> a`

Writing Type Generators

- Subclass of a BGEN-provided type generator class
- Override various methods that control generator, e.g.
  - how to declare an object of type X
  - how to convert from, to Python object
    - » using PyArg_ParseTuple() and By_BuildValue()
  - how to pass as InMode, InOutMode, OutMode argument
  - optional clean-up or error handling
- Can map single Python object to several C arguments
  - e.g. for variable-size buffers

Handling Buffer Arguments

- Buffer handling is different in Python than in C
- Example:
  - `read(fd, buffer, buflen) -> nbytes`
    translates into Python as
  - `read(fd, buflen) -> string_of_length_nbytes`
- Many different conventions exist in C
  - implicit length; length variable with InOutMode; etc.
- BGEN has generator classes for many schemes
  - or you can roll your own type generator
- Scanner needs hints to recognize buffer schemes

"Blacklisting"

The control module can "blacklist":

- specific functions
  - for dangerous or unneeded functions
- functions that receive or return specific types
  - for types that you don't know yet to handle

Advanced Features

- Error handling
  - designate a particular output variable as the error indicator
  - specify error test (could also test a global variable)
  - translates errors into exceptions
  - if no error, error variable can be ignored or returned
- Facilities for object type definitions
  - generates the usual boilerplate (typeobject initializer etc.)
  - can be used by type generators

Problems

Frankly, a lot:

- Customizing the header file scanner is a pain
  - Requires fairly consistent coding style in header files
  - Parameter transfer modes often guessed wrong
  - (Alternative: write generator calls by hand; also painful)
- Designing type generators is a black art
- No C++ support

Any Questions?